

# In Search of the Ideal Storage Configuration for Docker Containers

Vasily Tarasov<sup>1</sup>, Lukas Rupprecht<sup>1</sup>, Dimitris Skourtis<sup>1</sup>, Amit Warke<sup>1</sup>, Dean Hildebrand<sup>1</sup>  
Mohamed Mohamed<sup>1</sup>, Nagapramod Mandagere<sup>1</sup>, Wenji Li<sup>2</sup>, Raju Rangaswami<sup>3</sup>, Ming Zhao<sup>2</sup>  
<sup>1</sup>IBM Research—Almaden <sup>2</sup>Arizona State University <sup>3</sup>Florida International University

**Abstract**—Containers are a widely successful technology today popularized by Docker. Containers improve system utilization by increasing workload density. Docker containers enable seamless deployment of workloads across development, test, and production environments. Docker’s unique approach to data management, which involves frequent snapshot creation and removal, presents a new set of exciting challenges for storage systems. At the same time, storage management for Docker containers has remained largely unexplored with a dizzying array of solution choices and configuration options. In this paper we unravel the multi-faceted nature of Docker storage and demonstrate its impact on system and workload performance. As we uncover new properties of the popular Docker storage drivers, this is a sobering reminder that widespread use of new technologies can often precede their careful evaluation.

## I. INTRODUCTION

Operating Systems (OSs) use *processes* as a powerful and convenient computing abstraction. However, as the hardware capabilities of machines improved, it became cost-efficient to share the abundant hardware resources across multiple users and applications but with better isolation than native processes. Process *containers* [1] made this transition possible. The rapid adoption of containers is fueled by cloud computing whereby multiple tenants can transparently run their workloads on the same node. According to a recent poll, 25% of enterprises already use containers, while at least 62% are at some stage of adopting containers [2].

At its core, a container is a set of processes that is isolated from other processes or containers running in the system. Linux uses control groups (*cgroups*) [3], to limit the resource usage (e.g., memory and CPU) of containers, and *namespaces* [4] to confine process visibility. Since 2013, *Docker* has emerged as a composite technology that enables and simplifies the adoption of containers in modern OSs [5]. Docker containers allow users to effectively capture runtime environments in persistent *images* and easily execute the resident software inside dynamically created containers. Docker *images* contain all information needed to run the packaged software which significantly simplifies deployment across development, testing, and production environments.

While containers adequately address CPU and memory isolation across workloads, storage isolation is more challenging. At a fundamental level, storage for containers introduces the need to deal with an abundance of duplicate data referenced within container images. In a straw man implementation, Docker would need to create a complete copy of the image for

every running container. This would cause a great burden on the I/O subsystem and make container start time unacceptably high for many workloads. As a result, *copy-on-write* (CoW) storage and storage *snapshots* are popularly used and images are structured in *layers*. A layer consists of a set of files and layers with the same content can be shared across images, reducing the amount of storage required to run containers.

With Docker, one can choose *Aufs* [6], *Overlay2* [7], *Btrfs* [8], or device-mapper (*dm*) [9] as *storage drivers* which provide the required snapshotting and CoW capabilities for images. None of these solutions, however, were designed with Docker in mind and their effectiveness for Docker has not been systematically evaluated in the literature. With Docker, the depth of the file system software stack increases significantly. Besides, the available variety of CoW configurations and the possible diversity in workload mixes make the selection of the right storage solution challenging. A system architect must decide not only which CoW storage technology to use, but also how to configure it. For example, with *Aufs* and *Overlay2*, one needs to select the type of the underlying file system, and for device-mapper, which file system to format the thin-provisioned device with.

Unlike conventional workloads, Docker workloads induce frequent snapshot creation and destruction. The amount of data churn and the rate of image commits impact the behavior of the system significantly. Moreover, the number of layers in an image can impact performance both negatively and positively. Finer-grained images with more layers increase the amount of data that can be shared across different containers while on the other hand, more layers cause more overhead when accessing files in a container [10]. High-density containerized environments also increase the diversity of workloads and their parallelism. Resource isolation constraints imposed on individual containers can further morph workloads in non-trivial ways.

Although many questions arise when choosing storage for Docker containers, there is little to no guidance in peer-reviewed literature for selecting or designing storage for Docker [11]. In this study, we demonstrate the complexity of the Docker storage configuration and empirically demonstrate how the selection of the storage solution impacts system and workload performance. After discussing the relevant background on Docker and introducing the available storage drivers (§II), we make the following contributions:

- We present and discuss the different dimensions that

influence the choice of an appropriate storage solution for Docker containers (§III).

- We conduct a preliminary evaluation of the introduced dimensions and analyze their impact on performance and stability (§IV).

We found that, for example, for read-intensive workloads, `Aufs` and `Overlay2` are a good choice, while `Btrfs` can work well in deployments that experience a more diverse set of workloads. `Device-mapper` has a stable codebase but its performance is typically low and highly susceptible to the underlying storage speed. These and other observations can serve as a starting point for future work on storage management for Docker containers.

## II. DOCKER

Docker consists of a Command Line Tool (CLI) and a daemon (sometimes called engine) which continuously runs in the background of a dockerized system (Figure 1). The Docker daemon receives user commands from the CLI to build new or pull existing images, start or stop containers from images, delete containers or images, and other actions. Next, we introduce the relevant background on Docker images and its concept of storage drivers in more detail.

### A. Docker Images

Modern software relies heavily on the availability of a file system interface for its processes (after all, “Everything is a file” in UNIX-derived OSs [12]). Files not only store binaries, configuration files, and data, but also provide access to the system information and configuration (e.g., `/proc` and `/sys` file systems). Docker therefore dynamically creates a file system for a container to execute from. Docker file system *images* are similar to VM images, except that they consist of a series of *layers*. Every layer is a set of files. The layers get stacked with files in the upper layers superseding files in the layers below them. For example, in Figure 1, if the same file resides in layers  $L_0$  and  $L_1$ , then containers  $C1$  and  $C2$  only see the version from  $L_1$ . However, all files in  $L_0$  that do not exist in the upper layers will be seen by containers  $C1$  and  $C2$ . In almost all cases, a container’s file system is stored in the local storage device of the node in which it executes.

The number of layers in a single image ranges from one to several dozens [10]. Similarly to `git` [13], the layers are identified by fingerprints of their content. Different images often share layers, which provides significant space and I/O transfer savings. A layer in a Docker image often represents a layer in the corresponding software stack. For example, an image could consist of a Linux distribution layer, a libraries layer, a middleware layer, and an application layer.

A container image is read-only, with changes to its file system during execution stored separately. To create a container from an image, Docker creates an additional *writable* layer on top of the image with which the container interacts. When the container updates a file, the file is copied to the writable layer and only the copy is updated (copy-on-write). Unless the user

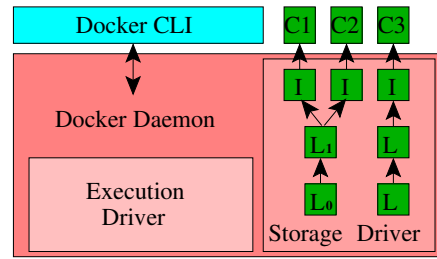


Fig. 1: Docker high-level design.  $C$  stands for Container,  $I$  for Image, and  $L$  for Layer. Three containers are created;  $C1$  and  $C2$  use images that share two layers.

saves the changes as a new layer (and hence a new image), the changes are discarded when the container is removed.

To store data persistently beyond the container removal, users can attach one or more file system *volumes* using a *volume driver* which provides the container access to data using protocols such as NFS and iSCSI. In this study, our focus is on challenges specific to configuring the local “ephemeral” file system for storing and accessing container images.

Users exchange images via a Docker *registry* service which typically runs on an independent machine. For example, Docker Hub is a popular registry service storing over 400,000 public images [14]. Docker clients cache images locally and therefore can start any number of containers from an image after pulling it only once. In this paper we assume that images are already pulled from the registry and focus on the startup and shutdown performance of containers.

Docker containers are often managed by high-level frameworks like Docker Swarm [15], Kubernetes [16], and others [17]. Furthermore, many products use Docker containers as a basic primitive for their workflows [18], [19]. In this paper, we generate workloads at the Docker level, not employing orchestration frameworks or complex workflows. We plan to extend our evaluation in future.

### B. Storage Drivers

Docker uses a variety of pluggable *storage drivers*<sup>1</sup> to manage the makeup and granularity of the layers and how changes to layers are saved. A storage driver is responsible for preparing a file system for a container. In this section, we briefly describe the available Docker storage drivers and their key differentiating features.

**VFS:** This simple driver does not save file updates separately from an image via CoW, but instead creates a complete copy of the image for each newly started container. It can therefore run on top of any file system. While this driver is not recommended for production due to its inefficiency, we discuss it here and in our evaluation since it is very stable and provides a good baseline.

**Aufs:** Another Union File System [6] is a union file system that takes multiple directories, referred to as *branches*,

<sup>1</sup>Storage drivers are sometimes also called *graphdrivers* because they maintain the graph (tree) of Docker layers and images.

and stacks them on top of each other to provide a single unified view at a single mount point. Aufs performs file-level CoW, storing updated versions of files in upper branches. To support Docker, each branch maps to an image layer [20].

To find a file, Aufs searches each layer/branch from top to bottom until the required file is found. Once found, a file can be read by the container. To update a file, Aufs first creates a copy of the file in the top writeable branch, and then updates the newly copied file. Deletes are handled by placing a *whiteout file* in the top writeable layer, which obscures all versions of the file in the lower layers. Aufs performance depends on many factors—application access pattern, the number of files, distribution of files across layers, and file sizes.

**Overlay and Overlay2:** Both of these drivers rely on the same underlying file system—OverlayFS [7]—which is yet another implementation of a union file system [21]. Unlike Aufs, OverlayFS is in the Linux kernel mainline and is therefore available in many Linux distributions out of the box. The Overlay driver was created for an earlier version of OverlayFS that supported only two layers: a read-only “lower” layer and a mutable “upper” layer. To merge several read-only layers the Overlay driver creates hardlinks to the shared files which can lead to inode exhaustion problems. The Overlay2 driver relies on the newer version of OverlayFS (kernel 4.0 and higher) which already supports 128 lower branches and therefore can merge up to 128 read-only layers without using hardlinks. In this paper we do not evaluate the older Overlay driver, and instead focus on the newer Overlay2 driver.

**Device-mapper (dm):** Unlike other storage drivers, dm operates at the block level instead of at the file system level. This driver uses Linux’s device-mapper subsystem to create *thin-provisioned* block devices. First, an administrator creates a *pool*, which typically stacks on top of two physical devices—one for user data and one for device-mapper metadata (e.g., block mappings). Second, when Docker creates a container, the dm driver allocates an individual *volume* for the container from the pool. To benefit from CoW, dm usually creates volumes as writable snapshots of previously created volumes.

However, for a Docker container to operate, it requires a file system, rather than a raw block device. So, as a third step, dm formats volumes with a configurable file system (usually Ext4 or XFS). By far the largest advantage of dm over Aufs and Overlay2 is that it can perform CoW at a granularity finer than a single file which is 512KB by default but configurable by the user. On the other hand, dm is completely file system-oblivious and therefore cannot benefit from using any file system information during snapshot creation.

**Btrfs:** Btrfs [8] is a modern CoW file system based on a *CoW-friendly* version of a B-tree [22]. Compared to Aufs and Overlay2, Btrfs is a file system that natively supports CoW and does not require an underlying file system. Btrfs implements the concept of *subvolumes* which are directory trees, represented in their own B-trees. Subvolumes can be efficiently snapshotted by creating a new root which points to the children of the existing root.

Storage driver	Space Efficiency	I/O perf	Driver Stability	Network Traffic
VFS	–	–	+	~
Aufs	~	~	–	~
Overlay2	~	~	~	~
dm	+	~	+	~
Btrfs	+	+	–	~
<b>Legend:</b>	+ positive		~ neutral	– negative

TABLE I: Gross comparison of container storage drivers.

The Btrfs storage driver [23] stores the base layer of an image as a separate subvolume and consecutive images are snapshots of their parent layer. Similar to dm, Btrfs performs CoW at the granularity of blocks which is more efficient in terms of performance and space utilization compared to file-based CoW. On the downside, Btrfs can experience higher fragmentation due to the finer-grained CoW.

### III. DIMENSIONS OF CONTAINER STORAGE

When designing a containerized environment, one has to choose from a large variety of storage options that span multiple dimensions, which we describe in this section.

**Storage drivers:** As discussed in Section II-B, Docker supports a variety of storage drivers. The storage driver choice is driven by three main considerations [24]: 1) space efficiency, 2) I/O performance, and 3) stability. Each driver uses a different approach to represent images and hence some drivers may not be suitable for certain types of workloads. For instance, if a large file is updated in a container, Aufs and OverlayFS would have to copy the entire file, decreasing performance and disk space usage. In another example, for containers with many files and deep directories, Aufs lookup operations can be slow because Aufs looks for a file at every layer of the image one at a time. Further, in our experience, some drivers are stable while others are experimental and not production-ready. All this makes choosing the appropriate driver difficult as it depends on the workload and the environment in which it is deployed. Table I summarizes the positive and negative impacts of each of the five supported Docker storage drivers across efficiency, performance, and stability metrics.

Table I also includes a network traffic metric to demonstrate that, unlike common belief, Docker exchanges images with a registry service at the file-level granularity no matter which storage driver is used. Significant improvements are possible in this area [25], [10].

**Image layers:** Docker’s representation of images as “layers of changes” adds another dimension of complexity. Having a large number of fine-grained, generic layers allows for a higher degree of sharing between different containers. This improves performance when pulling images from a central registry and reduces storage consumption at the Docker host. On the other hand, deeply layered images increase the latency of file system operations such as `open` or `read` [10], because more layers need to be traversed to reach the data. Depending on the specific user workload, the number of layers might or might not affect system performance.

**Storage devices:** Another important factor for Docker deployments is which type of storage devices should be used for storing container data. Docker is especially popular in the cloud and large cloud vendors such as Amazon, Microsoft, IBM, and Google offer a wide variety of storage options. For example, Amazon offers local ephemeral and remote persistent, block storage based on either SSDs or HDDs [26]. Depending on the VM instance type, one or the other (or both) can be selected. For remote storage, a variety of options with different throughput and IOPS values exist.

With Docker, the density and diversity of workloads running against the same storage device increase significantly. This is due to the fact that hosts can execute a larger number of lightweight containers compared to VMs. As a result, matching containers to the variety of available devices becomes a more challenging task. The burden is on the user to optimize for performance while keeping the cost low and ensuring that all workloads have sufficient resources to run smoothly.

**File systems:** While the Btrfs driver is based upon a Btrfs formatted device, the other drivers all include an additional file system layer. Overlay2 and Aufs are both union file systems that require an underlying file system such as Ext4 or XFS. dm on the other hand exposes a block device and hence, has to be formatted with a file system. While recommendations exist for some drivers regarding the choice of file system [27], it is unclear as to how this choice affects performance. Furthermore, every file system has a large number of parameters and their optimal values for Docker can be different than for traditional setups. For example, file system journaling is usually not needed as crash consistency is not required for a container’s ephemeral (local) storage.

**Workloads:** The “right” choices made with respect to the above dimensions are all influenced by the characteristics of the intended workload. The workload itself has several dimensions in terms of read/write ratio, the type of I/O (random or sequential), and the amount of data (size and number of files) that it operates on. A read-only workload that operates on a large number of files does not incur any overhead due to CoW but may under-perform with deeply layered images as it has to perform frequent lookups. Balancing such trade-offs is difficult, especially when the workload characteristics are not known or are not clearly defined.

Looking at the different dimensions, it becomes clear that picking the optimal storage solution for a containerized workload is not straightforward. While some guidelines exist [24], they do not cover all dimensions and do not provide clear evidence as to how the different aspects of a given deployment choice influence one another. Additionally, in many cases, the best choices are tightly coupled with the workload characteristics. Our evaluation study highlights some of the complexities of running workloads within the Docker storage ecosystem.

#### IV. EVALUATION

The evaluation presented in this paper is not intended as a comprehensive study. It is aligned with the goals of the paper

to demonstrate the impact (or the lack of it) of the Docker storage configuration on workload performance.

**Testbed:** In our experiments we used an IBM x3650 M4 server provisioned to support environments with a high density of containers per machine. Each node has two 8-core Intel Xeon 2.6GHz CPUs, 96GB of RAM, and a single 128GB SSD fully dedicated to Docker. The nodes runs a CentOS Linux 7.3.1611 (released in December 2016) with an updated kernel version 4.11.6 (released in June 2017) to enable support for Overlay2 and Aufs. For the same reason we updated Docker to version 17.03.1-ce (released in March 2017). We used Ext4 as the underlying file system for the storage drivers (except for the Btrfs driver). While we also ran experiments with XFS, our preliminary results indicate no significant difference caused by the file system and hence, we do not show the XFS results.

In our earlier experiments (not presented in here) we used an older kernel version (4.10) and Aufs hung under write-intensive workloads. Furthermore, when we switched to the newer 4.11 kernel, Aufs was returning errors on writes to the internal portions of large files on XFS. We reported this issue to the Aufs maintainers and it was recently fixed [28]. This contributed to our decision to mark Aufs stability as negative.

**Images:** We created several Docker images for the experiments. Each image is kept to the minimum size and only contains a Filebench [29] installation along with the required libraries. The total size of the image *without* the dataset is 2MB. Depending on the workload we created two images from the Filebench base image: a single-file image *Singlef* and a multi-file image *Multif*. The *Singlef* image contains a single 1GB file while *Multif* includes about 16,000 64KB-size files in a directory tree with an average depth of 3, similar to Filebench’s Web-server workload. The total size of the *Multif* dataset is therefore also 1GB.

**Workloads:** In all of our experiments we used the same high-level workload: we simultaneously started 64 Docker containers and then waited for all of them to exit. The idea behind using such a workload was to resemble a parallel job, such as a Spark job, which consists of a large number of equal small tasks. Each task is launched in its own container and operates on a different partition of the input data. We selected 64 containers assuming that every available core services 4 containers. We did not simulate application’s memory usage.

In each experiment, all containers were created from the same image and each container ran until it performed 600 I/O operations. The I/O speed was rate-limited to 10 synchronous IOPS per container, reflecting a scenario when a containerized application is not particularly I/O intensive. Though per-container I/O accesses are infrequent, all containers together produce a significant load on the I/O subsystem. In the case of no I/O congestion, every container would execute for 60 seconds. However, in reality, every container took longer to finish (up to 576 seconds) because 64 containers competed for the limited storage bandwidth.

For the *Singlef*-image-based containers we experimented with two workloads: 1) random 4KB reads from the 1GB file

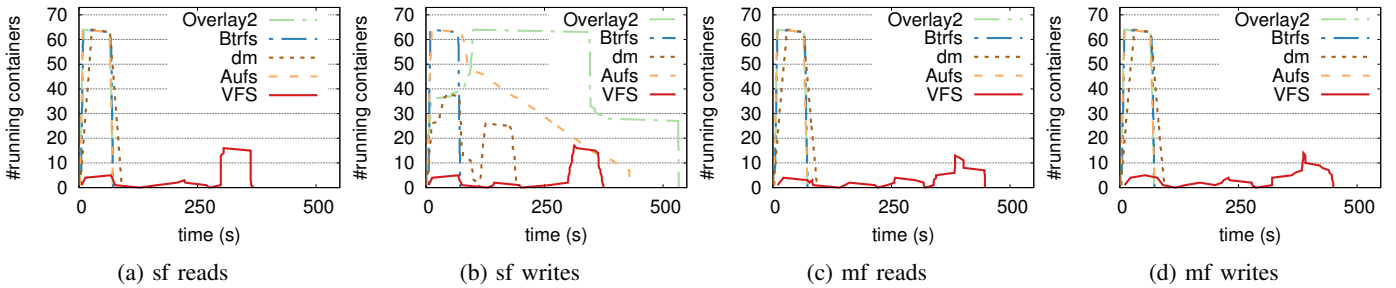


Fig. 2: Driver dimension: the number of running containers vs. time for different drivers and workloads.

(*sf-reads*) and 2) random 4KB updates to the file (*sf-writes*). The corresponding two workloads for the Multif-image-based containers are: 1) whole-file reads from randomly selected 64KB files (*mf reads*) and 2) whole-file overwrites of random 64KB files in the dataset (*mf writes*). For reproducibility, before starting an experiment we deleted all old containers, trimmed and reformatted the SSD, restarted the Docker daemon, and cleared file system caches.

Below we present three experiments, each exploring a different dimension: storage driver, number of layers, and storage device speed. Results for these experiments are presented in Figures 2, 3, and 4, respectively. We repeated every experiment three times and plot the average completion time and the minimum and maximum times as error bars for Figures 3 and 4. However, for the majority of the points, the deviation was too small to be visible on the graphs. Figure 2 contains a time-series graph where standard deviation is not applicable. We, however, verified that there were no significant variation between the results of different runs.

**Experiment 1—Drivers:** In this experiment we monitored how the total number of *running* containers changed over time for different storage drivers (see Figure 2). As is expected, for all drivers, the number of running containers first increased and then decreased again to zero by the end of the experiment, as all containers eventually finished. However, only Btrfs started containers fast enough to run all 64 containers at a high speed for *every* workload. Aufs and Overlay2 were as fast as Btrfs for all workloads except *sf-writes* as they need to perform expensive file-based CoW of the large 1GB file in this workload. The latency of the very first write in a container in this case reached up to hundreds of seconds due to the high I/O pressure. Interestingly, the starting time of containers for this workload was not impacted in the Aufs configuration, but was significantly increased (compared to other workloads) in the Overlay2 configuration. This resulted in a faster completion of the experiment with the Aufs driver.

In terms of completion time, the dm driver under-performed for the *sf-reads*, *mf-reads*, and *mf-writes* workloads by about 20%. The graphs indicate that this was mainly caused by the slower setup and destruction of the writable layer by device mapper. However, for the *sf-writes* workload, dm showed better completion times than Aufs and Overlay2. This is expected because dm’s CoW granularity is a block. We also

observe that dm’s performance took a more severe hit in *sf-writes* than in other workloads. We believe this is because dm has to first fetch 512KB blocks before updating them due to *read-before-write* [30] requirements.

For comparison, we also included the VFS driver in this experiment, which incurs unacceptably low performance. In fact, to avoid Docker timeouts and to be able to fit the resulting dataset on the SSD we had to limit the VFS driver experiments to 24 containers. When creating a writable layer for a container, the VFS driver copies all layers that belong to the corresponding image. The start of a Docker container requires the creation of two layers: one with configuration files specific to the container (e.g., `/etc/hosts`), and another one for the containerized application itself. Therefore, the start of a container results in copying 2GB of data, irrespective of the workload. This clearly indicates that storage CoW capability is crucial for Docker’s feasibility in practice. Because of its low performance, we do not evaluate the VFS driver further.

An interesting observation we made when running this experiment was that Overlay2 and Aufs copy a file when an application opens it with a `O_RDWR` flag, not when the first write is issued. By default, Filebench was opening files in `O_RDWR` mode, even for read-only workloads, which initially resulted in low performance for the read workloads with Overlay2 and Aufs. We modified Filebench to open files in `O_RDONLY` mode for read-only workloads. This experience shows that, first, applications have to be more careful when picking an open flag within containers. Second, if a containerized application performs both reads and writes, it is beneficial to segregate read-only data to one set of files, and write-only data to a different set of files, if possible. This will reduce the amount of data copied during the CoW process.

**Experiment 2—Layers:** Next, we evaluate the impact of the number of layers within an image on performance. For this experiment, we created layered versions of the Singlef and Multif images. For Singlef, each layer overwrites 1% of the 1GB file from a randomly chosen offset with random bytes. For Multif, each layer overwrites 1% of the 64KB files with random bytes. We varied the number of layers from 1 to 30 and measured the completion time for each of the four workloads (see Figure 3).

Previous work has shown that the number of layers does influence file access latencies [10]. Surprisingly, our results did not reveal any dependency between the performance and the



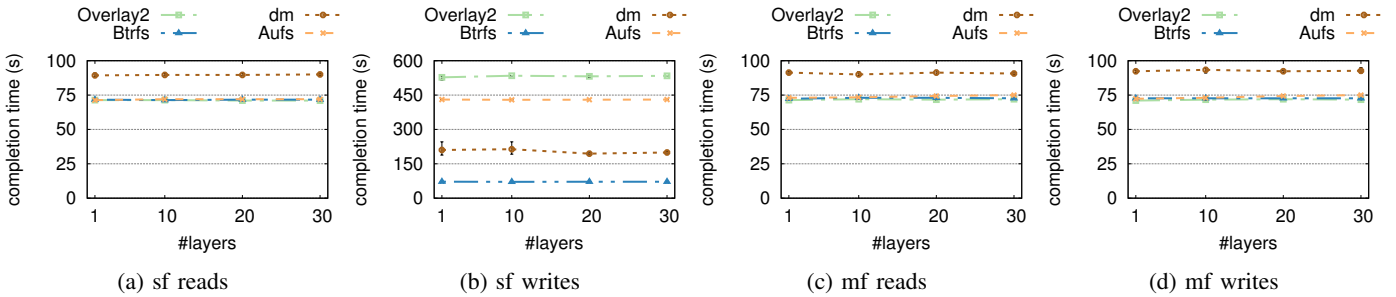


Fig. 3: Layer dimension: completion time vs. the number of layers for different drivers and workloads.

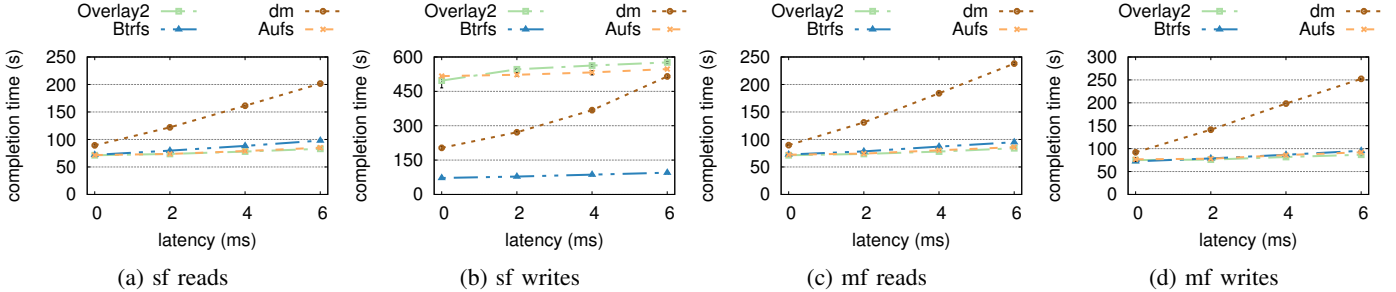


Fig. 4: Device dimension: completion time vs. device latency for different drivers and workloads.

number of layers. We experimented with higher percentages of inter-layer overwrites—5%, 10% and 20%—and still did not see any impact of the number of layers on performance. We believe that this is due to two reasons: 1) the overhead reported by [10] is only on the order of milliseconds and hence, does not have a significant impact for longer running workloads; 2) the overhead is only incurred for the first (uncached) access to a file which reduces its impact even further.

While this initial exploration does not indicate an impact of layers on performance, other interesting aspects of layering can be investigated such as its impact on storage utilization and caches. For example, we found that when a layered image is created on a client using a Dockerfile [31], the client does not fully benefit from the block-level CoW (in Btrfs and dm). E.g., the Singlef image with 10 layers was of 10GB size despite the fact that only 10MB of the file were changed in each layer. This happens because the image build process commits every layer, and every commit generates a layer at *file granularity* to exchange it with the Docker registry.

**Experiment 3—Devices:** To experiment with a wide range of devices that cloud providers offer to a consumer, we used our previously developed dm-delay tool to create virtual devices with arbitrary latencies [32], [33]. We extended dm-delay to support configurable queue depths and sub-millisecond latencies; we present here the results for 0, 2, 4, and 6 millisecond latencies when using a queue depth of 10. We deployed dm-delay on top of the SSD used in the previous experiments. Figure 4 highlights that in the majority of the cases, all drivers operate slower on slow devices. However, the impact when using the dm driver is much more severe than with file-system-based drivers. We think this is due to the fact that dm does not benefit from the page cache of the

underlying file system, as Aufs and Overlay2 do.

In this experiment we also occasionally ran into a problem with Btrfs. At times, under write-intensive workloads, Btrfs reported out-of-space errors even though there still was available space on the device. There are multiple reports on the web related to this issue [34], [35], [36]. This contributed to our decision to mark Btrfs stability as low in Table I.

## V. RELATED WORK

Docker provides some guidance on selecting a storage driver [24], but the recommendations are coarse grained and not supported with experimental results. There are also numerous non peer-reviewed postings online regarding container storage performance, but they tend to either not focus on storage [37] or they do not take the dimensions discussed in this paper into account [38], [39], [40], [41].

Studies exist that compare the performance of containers to virtual machines [42] or investigate container provisioning times [43]. However, they do not consider the different storage options that exist for Docker containers.

BetrFS improves the performance of copy-on-write file systems [44], but these optimizations have not been adapted to container workloads. While some additional work focuses on optimizing copy-on-write with containers [45], [46], it lacks a comprehensive evaluation and does not discuss the presented container storage dimensions.

Some researchers suggested to replace local storage drivers with distributed storage drivers [10], [47]. Slacker mixes NFS with server-side cloning to improve performance [10]. Shifter combines all storage layers into a single file stored in a parallel file system [47]. Our work applies to these approaches as well,

since the storage and file system are simply two dimensions that factor into container storage performance.

Finally, approaches exist to improve the management of container images. Exo-clones alter container image snapshots to be useable across different deployment environments [25]. CoMiCon introduces a distributed, cooperative registry that allows Docker hosts to share images between each other [48]. This is complementary to our work, since our focus is on performance and not management or storage efficiency.

## VI. CONCLUSIONS

Docker containers have become one of the major execution environments for many different applications, producing a unique workload blend that exercises storage's writable snapshot capabilities like never before. There is a variety of generic solutions that support CoW snapshots, but their effectiveness for Docker containers is not well understood. First, it is not trivial to understand which solution works best for Docker and in which situation. Second, the generic solutions were not designed with Docker in mind and may deliver suboptimal performance due to its unique I/O workload.

We demonstrated in this paper that Docker storage systems have many design dimensions and the design choices can impact Docker performance profoundly. At the same time, some dimensions that would, in theory, be expected to have a significant effect on performance (e.g., the number of layers in an image) do not have much impact in practice. While Btrfs seems to provide the best performance across the evaluated dimensions, its lower stability due to the out-of-space errors can make it prohibitive for production deployments. We hope that the information and experimental results presented here will draw the deserved attention to this new, exciting, and largely unexplored area.

**Acknowledgments:** We thank the anonymous AMLCS reviewers for their valuable comments. This work was supported in part by the NSF via grants CNS-1563883, CNS-1320426, CNS-1562837, and CNS-1619653.

## REFERENCES

- [1] P. Menage, "Adding Generic Process Containers to the Linux Kernel," in *Linux Symposium*, 2007.
- [2] 451 Research, "Application Containers Will Be a \$2.7Bn Market by 2020," [https://451research.com/images/Marketing/press\\_releases/Application-container-market-will-reach-2-7bn-in-2020\\_final\\_graphic.pdf](https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf).
- [3] "Control Group v2," <https://www.kernel.org/doc/Documentation/cgroup-v2.txt>.
- [4] E. Biederman, "Multiple Instances of the Global Linux Namespaces," in *Linux Symposium*, 2006.
- [5] "Docker," <https://www.docker.com/>.
- [6] "AUFS - Another Union Filesystem," <http://aufs.sourceforge.net>.
- [7] "Overlay Filesystem," <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>.
- [8] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, 2013.
- [9] "Device Mapper Thin Provisioning Targets," <https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt>.
- [10] T. Harter, B. Salmon, R. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [11] M. Lamourine, "Storage Options for Software Containers," *login: The USENIX Magazine*, vol. 40, no. 1, 2015.
- [12] "Everything is a File," [https://en.wikipedia.org/wiki/Everything\\_is\\_a\\_file](https://en.wikipedia.org/wiki/Everything_is_a_file).
- [13] "Git," <https://git-scm.com/>.
- [14] "Docker Hub," <https://hub.docker.com/>.
- [15] "Docker swarm mode ← Docker Documentation," <https://docs.docker.com/engine/swarm/>.
- [16] "Kubernetes," <https://www.kubernetes.io/>.
- [17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, 2016.
- [18] "Juju," <https://jujucharms.com/>.
- [19] "fabric8," <https://fabric8.io/>.
- [20] "Docker and AUFS in Practice ← Docker Storage Drivers ← User Guide ← Docker Engine ← Docker Documentation," <https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/>.
- [21] "Overlay with Multiple Lower Directory Support," <https://github.com/docker/docker/pull/22126>.
- [22] O. Rodeh, "B-trees, Shadowing, and Clones," *ACM Transactions on Storage (TOS)*, vol. 3, no. 4, 2008.
- [23] "Docker and Btrfs in Practice ← Docker Storage Drivers ← User Guide ← Docker Engine ← Docker Documentation," <https://docs.docker.com/engine/userguide/storagedriver/btrfs-driver/>.
- [24] "Select a Storage Driver ← Docker Storage Drivers ← User Guide ← Docker Engine ← Docker Documentation," <https://docs.docker.com/engine/userguide/storagedriver/selectadriver/>.
- [25] R. P. Spillane, W. Wang, L. Lu, M. Austruy, R. Rivera, and C. Karamanolis, "Exo-clones: Better Container Runtime Image Management Across the Clouds," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.
- [26] "Amazon EC2 Instance Types," <https://aws.amazon.com/ec2/instance-types/>.
- [27] "Docker and OverlayFS in Practice ← Docker Storage Drivers ← User Guide ← Docker Engine ← Docker Documentation," <https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/>.
- [28] "Aufs Returns ENOSUP when Writing in the Middle of a File on XFS," <https://sourceforge.net/p/aufs/mailman/aufs-users/thread/27641.1496877655%40jrobl/>.
- [29] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A Flexible Framework for File System Benchmarking," *login: The USENIX Magazine*, vol. 41, no. 1, 2016.
- [30] D. Campello, H. Lopez, L. Useche, R. Koller, and R. Rangaswami, "Non-blocking Writes to Files," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.
- [31] "Dockerfile Reference ← Docker Engine ← Docker Documentation," <https://docs.docker.com/engine/reference/builder/>.
- [32] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand, "A Fast and Slippery Slope for File Systems," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, 2016.
- [33] "Dm-delay," <https://www.kernel.org/doc/Documentation/device-mapper/delay.txt>.
- [34] "Ubuntu Thinks Btrfs Disk Is Full but It Is Not," <https://askubuntu.com/questions/464074/ubuntu-thinks-btrfs-disk-is-full-but-its-not>.
- [35] "Btrfs Out of Space, Even Though There Should Be 10% Left," <https://superuser.com/questions/1096658/btrfs-out-of-space-even-though-there-should-be-10-left>.
- [36] "Btrfs: No Space Left on Device," <https://nathantypanski.com/blog/2014-07-14-no-space-left.html>.
- [37] J. Eder and C. Murphy, "Performance Analysis of Docker on Red Hat Enterprise Linux 7 ← Red Hat Developers Blog," 2014, <https://developers.redhat.com/blog/2014/08/19/performance-analysis-docker-red-hat-enterprise-linux-7/>.
- [38] J. Eder, "Comprehensive Overview of Storage Scalability in Docker ← Red Hat Developers Blog," 2014, <https://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker/>.
- [39] P. Estes, "Storage Drivers in Docker: A Deep Dive," 2016, <https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/>.
- [40] R. Simon and T. von Eicken, "Perspectives on Docker," October 2014, <https://www.slideshare.net/rightscale/docker-meetup-40826948>.
- [41] W. Tyczynski, "1000 Nodes and Beyond: Updates to Kubernetes Performance and Scalability in 1.2 ← Kubernetes Blog," 2016, <http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-kubernetes-performance-and-scalability-in-12.html>.

- [42] P. Sharma, L. Chaufournier, P. J. Shenoy, and Y. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference (Middleware)*, 2016.
- [43] A. Hegde, R. Ghosh, T. Mukherjee, and V. Sharma, "SCoPe: A Decision System for Large Scale Container Provisioning Management," in *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD)*, 2016.
- [44] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuzmaul, and D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.
- [45] F. Zhao, K. Xu, and R. Shain, "Improving Copy-on-Write Performance in Container Storage Drivers," in *Storage Developers Conference*, 2016.
- [46] X. Wu, W. Wang, and S. Jiang, "TotalCOW: Unleash the Power of Copy-on-Write for Thin-Provisioned Containers," in *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, 2015.
- [47] R. S. Canon and D. Jacobsen, "Shifter: Containers for HPC," in *Cray User Group*, 2016.
- [48] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "CoMICon: A Co-Operative Management System for Docker Container Images," in *Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E)*, 2017.